PerfIso: Performance Isolation for Commercial Latency-Sensitive Services

Călin Iorgulescu* *EPFL*
Reza Azimi* *Brown University*
Youngjin Kwon* *U. Texas at Austin*
Sameh Elnikety *Microsoft Research*
Manoj Syamala *Microsoft Research*
Vivek Narasayya *Microsoft Research*
Herodotos Herodotou* *Cyprus University of Technology*
Paulo Tomita *Microsoft Bing*
Alex Chen *Microsoft Bing*
Jack Zhang *Microsoft Bing*
Junhua Wang *Microsoft Bing*

Abstract

Large commercial latency-sensitive services, such as web search, run on dedicated clusters provisioned for peak load to ensure responsiveness and tolerate data center outages. As a result, the average load is far lower than the peak load used for provisioning, leading to resource under-utilization. The idle resources can be used to run batch jobs, completing useful work and reducing overall data center provisioning costs. However, this is challenging in practice due to the complexity and stringent tail-latency requirements of latency-sensitive services. Left unmanaged, the competition for machine resources can lead to severe response-time degradation and unmet service-level objectives (SLOs).

This work describes *PerfIso*, a performance isolation framework which has been used for nearly three years in Microsoft Bing, a major search engine, to colocate batch jobs with production latency-sensitive services on over 90,000 servers. We discuss the design and implementation of *PerfIso*, and conduct an experimental evaluation in a production environment. We show that colocating CPU-intensive jobs with latency-sensitive services increases average CPU utilization from 21% to 66% for off-peak load without impacting tail latency.

1 Introduction

New server acquisition contributes to over half of the total cost of ownership (TCO) of modern data centers [8]. However, server utilization is low in data centers hosting large latency-sensitive services for two main reasons: First, latency-sensitive services are typically provisioned for the peak load, which occurs only for a fraction of the total running time [18]. Second, business-continuity plans dictate tolerating multiple major data center outages, such as tolerating the failure of two data centers out of three data centers within a continent while remaining capable of processing peak load. The high degree of over-provisioning is imperative: a livesite incident causing brief downtime results in lost revenue and frustrated users, while an extended downtime comes with negative headline news and irreparable business damage. Even slightly higher response times decrease user satisfaction and impact revenues [29, 10, 17].

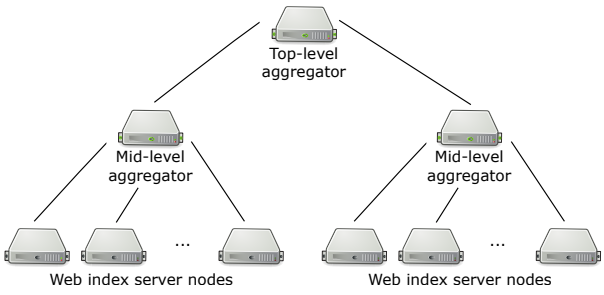


Figure 1: Architecture of index serving system of Web search engine with two aggregation levels (MLA and TLA). The user query is processed on index servers, which send responses to MLAs, which send aggregated responses to TLA.

Over-provisioning means that resource utilization is low, offering the opportunity to colocate batch jobs alongside latency-sensitive services [32, 18]. Colocation must be managed carefully lest it degrades performance due to competition on machine resources. Our main goal is to ensure that the end-to-end service-level objectives (SLOs) are met while increasing the work done by batch jobs. The main technical challenges arise from maintaining short tail latency (e.g., the 99^{th} latency percentile also called P99 latency) for the latency-sensitive services coupled with the complexity of commercial software and large deployments.

Oftentimes the service-level-objectives are not known explicitly for each individual component. For example, large commercial search engines contain tens of plat-

* Work done while authors were at Microsoft Research.

forms: to serve the web index, to build and update the web index, to manage user data and transaction history, to serve the most relevant advertisements, and to bill advertisers among many others. Modeling these components or assuming all their target latency values are known is not realistic.

Production environments are complex. A large data center comprising over 100,000 machines spans several generations of hardware. The generation gap can be up to 6 years, effectively constraining which hardware features can be used for performance isolation. Changes to the software stack running in a production environment are often infeasible. To be deployed on a large scale, the performance isolation framework must be robust, modular, and easy to debug. A good solution must provide the same performance guarantees seamlessly across all hardware and software configurations.

We describe our experiences in developing and deploying *PerfIso*, the performance isolation framework used in Microsoft's Bing clusters for over three years. We show how to colocate batch jobs with online services even when the tail response-latency requirements are within the order of milliseconds. We describe *CPU blind isolation* which dynamically restricts the cores that batch jobs use to protect the bursty interactive services even under high load. Depending on the load, batch jobs are given more or fewer resources to make progress.

Existing colocation approaches [20, 16, 34, 38] measure server-level performance metrics (e.g., query response times), and adjust resource allocation when the target is not met. This is not a good fit because if a query misses its target, it is already too late [17], and only end-to-end response time constraints are specified; per-layer service time limits are not.

We take a different approach: we ensure that there is always some slack in available resources such that abrupt changes in load do not impact response times. In contrast, traditional resource management polices focus on high resource-utilization while enforcing fair-sharing (most operating systems employ work-conserving scheduling algorithms). This works well for batch jobs, but does not account for factors such as the response-time latency of an interactive service. By using non-work-conserving resource management, we are able to adapt to changes in load and resource demands while treating the latency-sensitive service as a "black-box".

We focus on a concrete example: IndexServe — the Web index serving platform — because it is one of the largest in terms of machine count and has some of the strictest latency requirements. The web index is partitioned across hundreds or thousands of servers, and a user search query is processed in parallel on all servers. Responses are aggregated from the IndexServe machines on multiple levels (see Fig. 1). In such multi-layered systems, the slowest server dictates the response time [15].

To handle high load while meeting the strict latency requirements, many services are implemented as highly-optimized multi-threaded servers. The low query servicing-times make them highly bursty in nature: in several Bing services we find that, under high load, up to 15 threads become ready to run in just $5\mu s$. Due to the stringent tail-latency constraints, it is imperative to avoid scheduling delays, making the CPU the main bottleneck in our approach. We show that statically restricting CPU cores or CPU cycles does not fully solve the problem and fails to take advantage of idle cores during off-peak.

Our key goal is to ensure that interactive services perform equally well with batch jobs colocated as when they run alone. We show that CPU blind isolation successfully protects IndexServe while increasing average CPU utilization from 21% to 66% by colocating it with CPU-intensive jobs.

The main contributions of this work are as follows:

i. Identifying the key challenges of colocating batch jobs with large production latency-sensitive services and analyzing the effectiveness of operating system mechanisms to monitor and control resources.

ii. Introducing *CPU Blind Isolation* — a technique to mitigate harmful CPU-level interference between tenants.

iii. Designing and implementing the *PerfIso* performance isolation framework which allows batch jobs to be run alongside latency-sensitive services without any tail latency degradation.

iv. Evaluating *PerfIso* on a single machine to compare it to other alternatives, and on a 75-node production cluster, both running a real-world commercial online interactive service.

2 Background

We refer to the latency-sensitive user-facing services as *primary tenants*. All resources of a machine need to be available for them, since they generate the revenue that pays for the actual machines. The main goal of our system is to colocate batch jobs with a latency-sensitive user-facing service without impacting its response times. Thus, the primary always runs unrestricted and unmodified.

Batch jobs that run on these machines are *secondary tenants* and are treated in a best-effort manner — any resources they use are released to the system whenever the *primary* needs them. If the primary does not utilize all available resources, the secondary will be allowed to use some of them.

2.1 Applications and Services

Primary tenants comprise the latency-sensitive services which are governed by strict response-time SLOs. They are characterized by the following:

1. **a complex layered architecture** – hard to model or predict, since responses are computed in parallel and then aggregated.
2. **short tail latency** – any layer can severely impact query response times.
3. **highly bursty nature** – the service frequently spawns a large number of workers in a short period of time (order of microseconds).

Example primary tenant. We take the IndexServe component of **Bing** search as an example. IndexServe receives user queries and fetches potential matches from a search index. It can perform a variety of lookup and ranking operations. Its response times are within the order of milliseconds, and SLOs dictate that the 99^{th} percentile must stay within a 1-millisecond limit of its expected value (i.e., without colocation). Fig. 1 shows a simplified description of the layered architecture of IndexServe. Our measurements indicated that during a time frame of $5\mu s$, up to 15 threads became ready for execution.

Example secondary tenants. Non-latency-sensitive, big-data applications run alongside the primary. Popular big-data frameworks such as Hadoop [1], YARN [31], Apache Spark [35, 7], or Apache Flink [9] allow running a wide-range of compute-intensive (e.g., machine learning), and disk-bound (e.g., search index preparation and aggregation) jobs. Additionally, each server needs to run an *HDFS DataNode* process (for data replication), and a *YARN NodeManager* process (to handle individual task creation/destruction).

Both classes of tenants require access to several resources: CPU, disk, memory, network, etc. Accounting for potential resource bottlenecks is paramount in maintaining the performance of the primary. However, that alone is not sufficient, as our system needs to ensure that the secondary can make adequate progress, increasing the amount of work done when the primary is underutilized.

2.2 Driving Forces and Constraints

We focus on the performance requirements of the primary, without making any assumptions about its implementation. This enables wide-spread deployment, but makes it difficult to identify when the secondary interferes with the primary. Although we consider the CPU the main bottleneck, other resources also need to be monitored for contention.

Another key aspect is controlling resource access. Most operating systems already offer static mechanisms to restrict or prioritize access to a certain resource. While comprehensive, these are insufficient when dealing with bursty latency-sensitive workloads.

Given the complexity of production systems, it is hard to change the primary or the operating system (especially the kernel) due to the high costs of development, testing and deployment. Rather, our solution relies on features readily-available, and makes very few assumptions about the primary workload. In a nutshell, we treat the primary and the OS as a "black-box".

3 System Design

3.1 CPU Blind Isolation

CPU is a prevalent bottleneck resource for most low-latency services and big-data frameworks [27]. Modern OSes implement effective means to statically manage CPU time across tenants [3, 4], such as throttling CPU cycles or restricting CPU cores. However, in Section 6.1.4 we show that they are ineffective because they cannot automatically adapt to the bursty workloads.

We propose a new technique called *CPU Blind Isolation*, which restricts which cores secondary tenants use based on core utilization information read from the OS. The key idea is to ensure that the primary *always has some headroom* (i.e., **buffer idle cores**), to absorb any primary worker-threads that wake up. The number of buffer cores is computed after offline-profiling of the primary using a sufficiently-heavy workload.

As the primary must always run unrestricted, we *restrict the secondary* to run only on a subset of cores. The secondary is allocated the cores remaining after subtracting the cores used by the primary and the number of buffer idle cores.

For example, consider a machine with 48 (physical) cores running a primary that needs 4 buffer cores to absorb bursts. If the primary uses 20 cores, the secondary would be restricted to 24 cores. If the primary goes up to 24 cores, the secondary is immediately restricted to 20.

Why not change the OS scheduler? We recognize that this solution can be implemented at the scheduler level. We argue that this is impractical and imposes significant overhead in large-scale deployment. Bugs introduced to the scheduler by seemingly-trivial changes are laborious to track down and can cause unexpected performance degradation. For example, the well-established Linux Completely-Fair-Scheduler has been found to have had bugs which caused threads to wait even when idle cores were available [21]. These bugs persisted in the codebase for several years. Our approach achieves performance isolation without interfering with the scheduler or the scheduling policy.

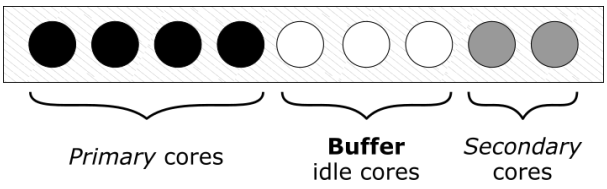


Figure 2: Conceptual representation of *CPU blind isolation*. The primary is unrestricted, and can run on any core, while the secondary is restricted to a subset of cores such that the primary always has a buffer of idle cores.

Non-work conserving scheduling In contrast to most OS schedulers, *blind isolation* is non-work conserving by nature. It deliberately chooses to leave several cores idle in order to properly measure and react to changes in the amount of work done by the primary. It is known that non-work conserving schedulers can improve performance in multi-processor scenarios [13]. We find that, similarly, blind isolation helps improve CPU utilization in the case of colocation.

3.1.1 Counting Idle Cores

An important requirement of our solution is a *low-latency*, *low-overhead* means of obtaining CPU utilization information. More specifically, we need to know how many cores are idle. We consider a core to be idle if the *idle thread* is running there.

The Windows scheduler keeps track of idle cores and provides this information through a system call. This system call returns a bit mask with the bits corresponding to the idle CPUs' ids set. We tested several other approaches relying on different metrics (e.g., recorded idle times, counting active threads), but found this solution to be best in terms of latency, overhead, and accuracy.

3.1.2 Allocating Cores to the Secondary

Once we know how many cores are idle we can detect whether the secondary needs to give up cores to the primary, or if the primary is not using all available cores. We assume that the secondary is CPU-intensive, and thus will fully occupy all cores allocated to it. If **I** is the number of idle cores in the system, **B** is the number of buffer cores, and **S** is the number of cores allocated to the secondary, then: if $\mathbf{I} < \mathbf{B}$, **S** is decreased, and if $\mathbf{I} > \mathbf{B}$, **S** is increased.

3.2 Managing Other Resources

Disk. We *choose which disks* are best suited for the primary and secondary. We find that it is necessary to separate the disks which are on the critical path of the primary from those used by the secondary. This is motivated by the nature of the tenants: the primary is highly-tuned towards read-only random accesses, so it is assigned to a striped set of solid-state disks (SSDs). In contrast, batch jobs often perform both reads and writes and mostly sequential in nature, so they are assigned to a striped set of hard-disks (HDDs).

Memory. Most low-latency services will manage their caches explicitly, loading data from disk as necessary depending on incoming queries. Furthermore, primary services are engineered to have a fixed working set and a stable memory footprint. We cannot compromise on this, and must guarantee the primary's ability to make full use of the memory. This is achieved by *limiting the memory footprint* of the secondary. When memory runs very low, secondary processes are killed.

Egress network packets. We *throttle* the outbound traffic of the secondary, marking it as low-priority and allowing the primary to maintain its throughput and response latency. This prevents the secondary from affecting the responsiveness of the primary.

4 Implementation and Deployment

We implemented *PerfIso* as a user-mode service based on the techniques and OS mechanisms described. Most of the static limits that *PerfIso* enforces are read from cluster-wide configuration files distributed through the Autopilot [14] environment. The resource limits can be altered independently at runtime by issuing a command to *PerfIso*. A client-application can also be used locally for debugging.

Although it is possible to obtain the unique process identifiers (PIDs) of the secondary tenants, Autopilot eases this task by keeping a list of running services and their respective information. Each secondary tenant process is placed in a unified *Job Object* configured dynamically by *PerfIso*.

4.1 Isolation Algorithm

The dynamic limits set by *PerfIso* need to be adjusted often. The state of the system needs to be read and the control knobs updated accordingly. Polling is important because the state of the primary can change quickly. Unfortunately, constantly updating certain settings can become harmful to the performance of all services. Thus, *polling and updating are separated* in *PerfIso*. We poll utilization data (e.g., CPU) continuously in a tight loop and we update the dynamic limits of the system on-demand based on the measured change in resource requirements.

Choosing the number of buffer cores. *CPU blind isolation* uses buffer cores to ensure that tail latency is protected while the system adjusts to changes in load. This

requires that sufficient buffer cores are allocated to absorb bursty workloads. A one-off measurement of the primary under its provisioned peak load is needed to find how many threads can become ready for execution.

We evaluate different buffer core values in Section 6.1.3, and find that 8 cores are enough for IndexServe to maintain its 99th percentile SLO on our servers.

I/O throttling. The monitoring mechanisms provide only per-device I/O statistics, without discerning which processes originated the operations. In order to provide per-process throttling of I/O, we use *Deficit-Weighted-Round-Robin* [19]. Each process is assigned an I/O priority and one or more limits that need to be enforced (e.g., bandwidth, IOPS). Based on its priority, each process is assigned a weight – the higher the priority, the larger the weight. We then measure the number of completed I/O requests per second (or IOPS) per drive, and use a moving average.

We compute the portion of the requests a given process is responsible for based on its weight. Considering w_i^t the weight of process i at time t, and $curr^t$ the IOPS value measured at time t, then the demand of process i is:

$$D_i^t = \sum_{t'=t-\Delta}^{t} \frac{w_i^{t'} \times curr^{t'}}{\sum_{\forall j} w_j^{t'}}$$

We mark the lower limit of process i with lim_i, which represents the minimum amount of IOPS that process i is guaranteed. The deficit of this process with regard to the limit is:

$$\mathrm{Def}_i^t = \frac{curr^t - min(lim_i, D_i^t)}{min(lim_i, D_i^t)}$$

The I/O priorities of processes are adjusted based on the computed deficit values.

4.2 Deployment in Production Clusters

All machines run under a management framework such as Autopilot [14]. This provides machine wiping, imaging, backup, and monitoring functionality. Autopilot provides a stable service management interface to start, stop, and configure software. *PerfIso* is run as an additional service in Autopilot, making it easy to deploy, and to configure across various different environments.

PerfIso is designed to have a "kill-switch", so that it can be quickly deactivated. This is useful when debugging production issues, and it allows quickly excluding *PerfIso* as a potential cause.

PerfIso is fully recoverable, since all parameters are stored in the cluster-wide configuration files. In the event of a crash, Autopilot will bring it up again, and *PerfIso* will resume its function by loading its state from disk.

PerfIso ensures that its settings do not affect those employed by the primary. For example, if the primary uses core affinitization for performance reasons, then *PerfIso* would not override these settings when attempting to accommodate the secondary.

5 Experimental Evaluation

5.1 Objectives

1. How is tail latency impacted by colocating batch jobs with the primary without *PerfIso*?
2. How effective is *PerfIso* in maintaining tail latency when a batch job is colocated with the primary?
3. How does CPU *blind isolation* compare to static isolation mechanisms provided by modern OSes?

5.2 Machine Configuration

We evaluate our solution on typical production hardware. Each server has two Intel Xeon E5-2673 v3 processors with 12 physical cores per die (a total of 48 cores with hyper-threading), 128 GB of RAM, and a 10 GbE Ethernet card. Storage is provided by 2 striped volumes: 4× 500 GB SSD drives, and 4× 2 TB HDD drives. The servers run Microsoft Windows Server 2016.

5.3 Experiment Setup

We use Bing IndexServe as the **primary** tenant in our experiments. IndexServe processes a search query to find a match using a large index partitioned across machines and replicated for performance.

IndexServe is setup with an index slice of 569 GB, and uses approximatively 110 GB of memory to cache recently accessed web index data. The index slice is stored on the striped SSD volume which IndexServe uses exclusively. IndexServe relies on the SSDs' low I/O latency to maintain its tail latency requirements. The HDD volume is only used by IndexServe for logging, being shared with the secondary. The service is configured to return the most relevant matches.

Primary workload. We use a trace of 500k real-world queries from early 2017 to put load on the primary. A separate client machine is used to submit queries from the trace. We first replay a warm-up trace of 100k queries at a rate of 300 queries / second (QPS) so that IndexServe ramps up and reaches a steady-state. The warm-up is not reported as part of our measurements.

We vary the load by changing the query arrival rate, *i.e.*, we replay our trace at different query rates. The following represent a reasonable approximation of query arrival rates that an IndexServe machine might receive at the time that this paper was written:

- 2,000 QPS - approximating average load
- 4,000 QPS - approximating peak load

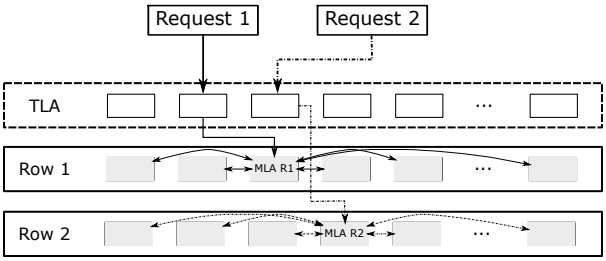


Figure 3: Request processing on the 75 machine IndexServe cluster described in Section 5.3. All gray machines run IndexServe and hold a slice of the index.

The client application replays the query trace in an open loop and sends queries according to a Poisson process distribution.

Secondary workload. We use a synthetic micro-benchmark as the **secondary** to stress the CPU by actively utilizing as many CPU cycles as the system permits, pushing *blind isolation* to its limits. This **CPU bully** is a multi-threaded program with each worker thread computing the sum of several integer values. The number of worker threads is configurable and we vary it up to the total number of logical cores present on the system. The bully maximizes CPU utilization since there are very few memory or external storage accesses.

Single-machine experiments. We run IndexServe on a single machine configured as described above. We measure the impact of CPU contention on tail query response time, the most important metric being the 99th percentile.

Cluster experiments. We setup IndexServe across 75 machines, in the following manner: the index is split into 22 partitions (or *columns*), and each column is replicated by a factor of 2 (total of 2 *rows*). Each IndexServe server holds a partition of the index similarly to the single-box runs. The top-level aggregator (TLA) runs on 31 separate machines than the ones that hold the index. The mid-level aggregator (MLA) runs on IndexServe machines, and each request may get forwarded to a different MLA based on the TLA load-balancing. Fig. 3 shows an example of the system processing 2 incoming requests.

Each IndexServe machine also runs an HDFS client because many batch jobs that are used in production run on top of frameworks such as Hadoop and, thus, rely on HDFS for storage access. In addition to other experiment-specific *PerfIso* settings, we also set the following static disk bandwidth limits: data replication is limited to 20MB/s, and HDFS clients are limited to 60 MB/s. All I/O operations done by HDFS are unbuffered.

A client is setup on a separate machine and configured to submit queries to the TLA machines. We then run a trace of 200k queries at a total rate of 8,000 QPS. The TLAs will load-balance these queries across the 2 rows, resulting in an average workload of 4,000 QPS for each IndexServe machine.

Additionally, we use a **Disk bully** to ensure that I/O generated by HDFS does not cause any server to straggle. We setup DiskSPD [5] to create an I/O bound workload on the HDD strip of each machine. We perform a mixed read-write workload, with 33% reads and 67% writes, with sequential accesses and synchronous I/O operations.

6 Experimental Results

We first evaluate *PerfIso* on a single machine and measure the effectiveness of CPU blind isolation. We then move on to a 75-machine cluster and analyze CPU isolation mechanisms, measuring latency end-to-end and at each component level. The main metric used is the 99th percentile of query response latency.

6.1 Single-machine Experiments

Going further, we analyze the baseline (or standalone) behaviour of IndexServe and three colocation scenarios:

- **No isolation** – The primary and secondary are colocated without any isolation.
- **Blind isolation** – The secondary is dynamically restricted in terms of CPU cores using our technique.
- **Alternative isolation** – The secondary is successively restricted in terms of CPU cores, and CPU cycles using OS-specific mechanisms.

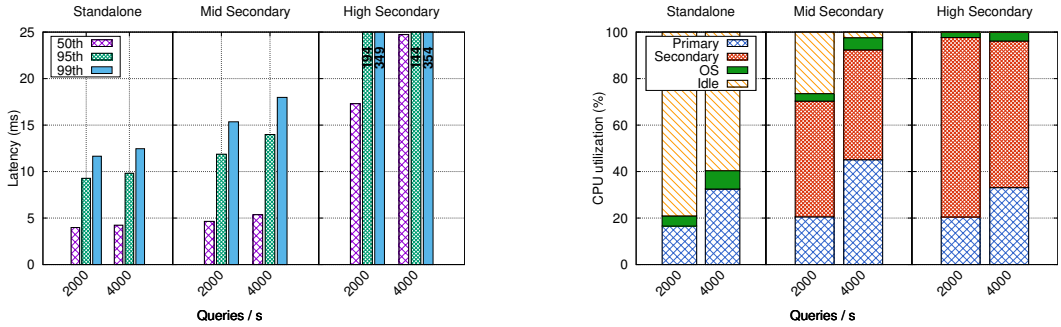
Our goal is to also maximize the amount of work done by the secondary, so we first configure each isolation technique with "relaxed" settings. We then successively restrict the secondary until either the SLO is met, or until the secondary no longer gets any work done.

6.1.1 Baseline

First, we measure how IndexServe performs when it runs standalone (i.e., no colocation). The 1st bar groups of Figs. 4a and 4b report the query latency and CPU utilization, respectively. The median query time is 4ms, and the 99th percentile is 12ms, both for 2,000 and 4,000 QPS. The average CPU utilization is low, with the CPU remaining idle for 80% and 60% of time, respectively.

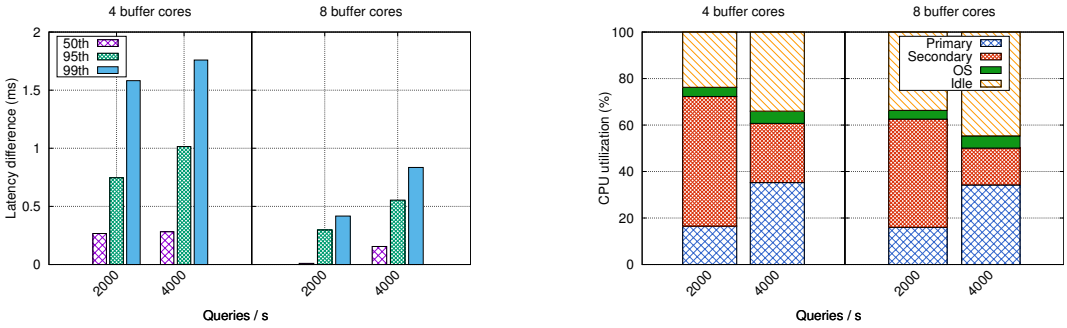
6.1.2 No Isolation

We colocate the primary and secondary, configuring the bully to use either *mid* (24 threads) or *high* (48 threads). The 2nd and 3rd columns of Figs. 4a and 4b report query latency and CPU utilization for the *mid* and



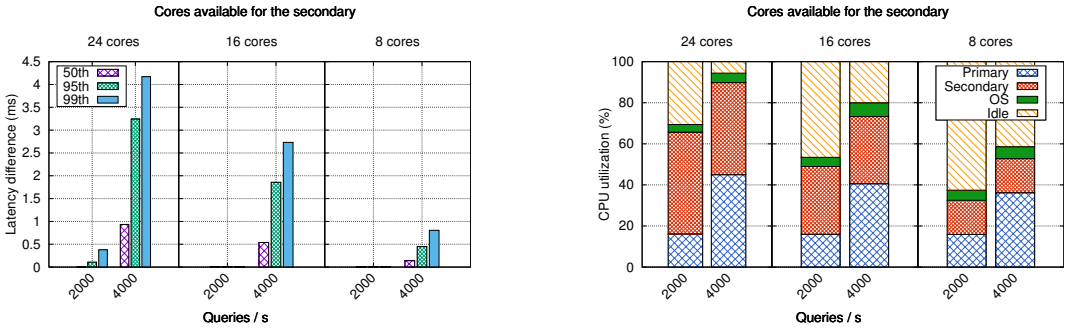
(a) Query response latency (b) CPU utilization

Figure 4: Single machine run of IndexServe standalone (no colocation) vs. colocated with an unrestricted secondary. A *mid* secondary increases the 99th percentile query latency by up to 42%, while a *high* increases same by up to 29×.



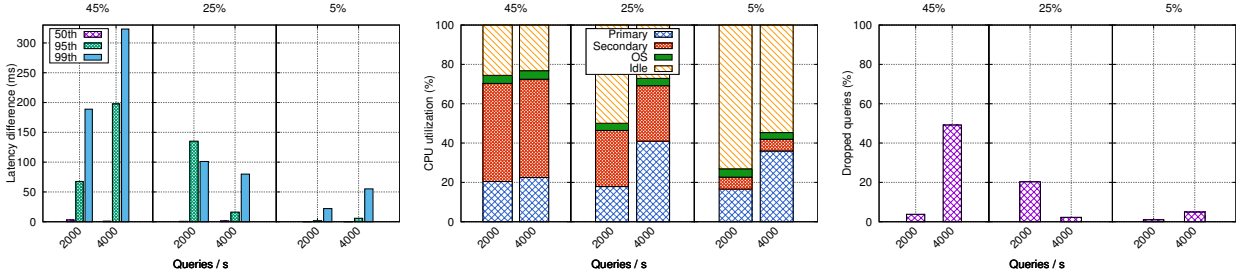
(a) Query latency degradation (b) CPU utilization

Figure 5: Single machine run of IndexServe colocated with a secondary restricted using blind isolation. Using a buffer of 8 CPU cores, the 99th percentile query latency is less than 1 ms off from the standalone case.



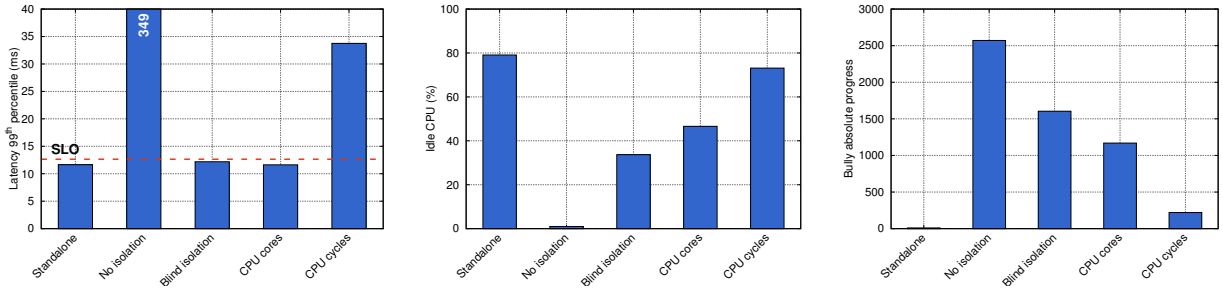
(a) Query latency degradation (b) CPU utilization

Figure 6: Single machine run of IndexServe colocated with a secondary with CPU cores statically restricted.



(a) Query latency degradation (b) CPU utilization (c) Dropped queries

Figure 7: Single machine run of IndexServe colocated with a secondary with CPU cycles statically restricted.



(a) Query latency (b) Idle CPU (c) Secondary progress

Figure 8: Comparison of isolation approaches on a single machine run of IndexServe at 2,000 QPS colocated with a secondary running in *high*-mode. CPU blind isolation uses 8 buffer cores, CPU cores allows the secondary to use 8 cores, and CPU cycles restricts the secondary to 5% of CPU time.

high configurations, respectively. The impact of colocation is substantial. The *mid* case reaches 15ms and 18ms in the 99th percentile for 2,000 and 4,000 QPS (higher than the baseline by 3-5ms). For *high*, these values reach 349ms and 354ms (a 29× degradation). Between 11% and 32% of queries timeout.

Fig. 4b shows the CPU utilization for the primary and secondary, and idle CPU. Interestingly, when colocated with the *mid* secondary, the CPU utilization of the primary increases up to 40% — IndexServe tries to compensate for the increase in pending queries by starting more workers. While this successfully prevents dropped queries, it ultimately aggravates CPU contention, and the latency SLO is not met. Another consequence is that the secondary gets less CPU time overall, since the primary will push it out from the only cores where it can run.

In the case of the *high* secondary, more than 32% of queries submitted at *peak* primary load are dropped due to the longer processing times, causing a decrease in primary CPU utilization.

6.1.3 Blind Isolation

We further use the 48 worker variant (*high*) secondary to evaluate the efficiency of isolation mechanisms.

We evaluate the blind isolation mechanism by reserving 4 and 8 buffer logical-cores, respectively. The insight here is to allow the primary to have a buffer of cores to start new worker threads when load increases.

Fig. 5a and Fig. 5b report our findings in terms of latency degradation and CPU utilization. We find that provisioning 8 idle logical cores is enough to ensure less than 1ms of degradation for the 99th latency percentile of the *high* workload.

6.1.4 Comparison to Alternative Isolation Methods

We next analyze the effectiveness of two common methods of static CPU resource management which are available in most modern OSes: *restricting CPU cores* and *restricting CPU cycles*. Windows provides these mechanisms through the *Job Object* abstraction [3], while Linux does so through the *cgroups* framework [4].

Restricting CPU cores. We successively restrict the secondary to use only 24, 16, and 8 cores of all 48 available logical cores. The primary is guaranteed exclusive and unimpeded access to the remainder, but can also compete for the secondary's cores. Fig. 6a shows the degradation of query response latency for each case.

Fig. 6b shows the overall CPU utilization breakdown when the bully is restricted to a subset of cores. When IndexServe is under average load, the secondary can claim up to 33% of the CPU time. While this is an important gain, the servers need to be provisioned for *peak* load, thereby reducing the subset of cores allocated to the secondary to 8 cores. With IndexServe under *peak* load, the secondary can only use up to 17% of the CPU time.

Restricting CPU cycles. We successively restrict the secondary to 45%, 25% and 5% of the overall CPU time.

Fig. 7a reports the measured degradation of latency, and Fig. 7c shows the percentage of queries that were dropped (because of increased processing times). Giving the bully even as little as 5% of CPU time still produces degradation. Furthermore, as opposed to restricting CPU cores, there is always some percentage of queries that get dropped, ranging from 50% to around 1% (in the best case). Fig. 7b shows that using this method less CPU time goes to the secondary.

The main reason this technique yields results worse than restricting CPU cores is that multi-threaded services such as IndexServe launch short-lived worker threads to process incoming requests. If these threads end up being queued for execution instead of being launched right away, it creates a cascading effect which impacts all incoming queries. Despite the secondary not utilizing more than its share of CPU time, IndexServe worker threads

get delayed, leading to considerable degradation.

Progress of the secondary. Finally, we analyze the amount of work that the secondary gets done under isolation, as a percentage of the total work done when unrestricted. We report for each IndexServe workload the point where latency degradation was lowest for that experiment. Blind Isolation allows the secondary to achieve 62% and 25% of the work it did unrestricted, for 2,000 and 4,000 QPS, respectively. Restricting CPU cores yields a more modest 45% and a similar 30%, respectively. Restricting CPU cycles fares worst, yielding only 9% in both cases.

6.2 Cluster Experiments

We next look at how *PerfIso* performs in a production cluster. We evaluate at an approximation of peak load to stress the system, attempting to impact tail latency. This makes *PerfIso* throttle the secondary more aggressively to accommodate the primary.

As described in Fig. 3, requests arrive at one of many top-level aggregator (TLA) machines, which forwards the request in a round-robin fashion to one of the two rows of IndexServe machines (each row holds a partitioned copy of the search index). The TLA chooses an IndexServe machine from the row to act as the mid-level aggregator (MLA) for a particular request. The MLA queries all the other IndexServe instances in its row (including the local one), aggregates all results, and formulates the final query response.

We run each experiment 8 times, and measure query latency at a) each server, b) at each layer, and c) end-to-end. Fig. 9a reports the baseline query response latency, averaged across IndexServe machines, across MLAs, and across TLAs. The HDFS client takes up to 5% of total CPU time.

We start our CPU bully and configure *PerfIso* on each IndexServe machine for blind isolation in the same manner as the singlebox runs. Fig. 9b shows the query response latency for this CPU-bound workload. Compared to the baseline, the 99th percentile reported by IndexServe, MLA, and TLA instances increases by at most 0.8, 0.4, and 1.1 milliseconds, respectively.

We configure *PerfIso* to throttle disk I/O to either 100MB/s, or 20 IOPS/s, for a given operation data chunk size of 8KB. Fig. 9c shows the query response latency for this Disk-bound workload. Compared to the baseline, the 99th percentile reported by IndexServe, MLA, and TLA instances increases by at most 0.8, 1.2, and 1.1 milliseconds, respectively.

Progress results with larger cluster. Finally, we show production results for a cluster of 650 IndexServe machines processing live user queries while colocated with a large batch job executing the training phase of a machine-learning computation. Fig. 10 shows key performance metrics: load in QPS, 99th percentile latency of query response times measured at the TLA, and server CPU utilization averaged across all machines. Importantly, CPU utilization averages 70% over 1 hour.

6.3 Takeaways

We now present a sum-up of our evaluation, referring to the objectives established in Section 5.1:

1. Fig. 8a shows that a CPU-bound batch job can importantly affect the 99th percentile query latency of the primary, reaching up to 29$\times$ degradation. Fig. 4a shows that even a mildly CPU-intensive job can cause interference and can increase tail latency by up to 42%.

2. Fig. 8a shows that *blind isolation* successfully protects tail latency under medium load (2,000 QPS), and Fig. 4a shows that this holds for *peak* loads (4,000 QPS) as well. In the latter case, the 99th percentile is within 1 ms of the standalone case. Fig. 9 reports the results for 8 runs of an approximated *peak* load (4,000 QPS) on a production cluster, showing that *PerfIso* successfully protects tail latency. The query response latency tail for CPU and Disk-bound jobs (Figs. 9b and 9c) is within 1.2 milliseconds of the standalone case (Fig. 9a). Fig. 10 shows that *blind isolation* raises CPU utilization to 70% through colocation over the course of 1 hour on a 650-machine IndexServe cluster.

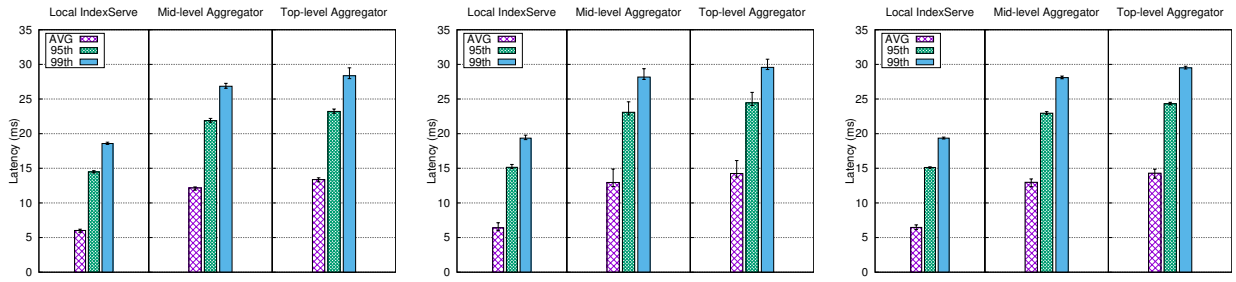
3. Fig. 8 reports a full comparison of all our evaluated techniques, showing that *blind isolation* and *cpu cores* both protect tail latency. However *blind isolation* manages to reduce idle cpu time by a further 13% compared to *cpu cores*, and allows the secondary to perform 17% more work. *CPU cycles* fails to protect tail latency.

We conclude that *PerfIso* successfully protects tail latency across all IndexServe machines, ultimately preserving the end-to-end SLOs.

7 Related Work

Many existing solutions propose colocation to increase data center utilization, but rely on information about the primary tenant's SLO and workload, or on specific hardware support. The complexity and performance characteristics (e.g., tail latency requirements and bursty nature) of primary tenants pushed our design into a different direction, adopting a black-box model with few assumptions on hardware for large-scale deployment.

MS Manners [12] provides CPU-level performance isolation on single-core machines by restricting the CPU



(a) Standalone (b) CPU-bound secondary (c) Disk-bound secondary

Figure 9: Latency values for IndexServe running on a production cluster: Fig. 9a shows the baseline, Figs. 9b and 9c show the result of colocation with CPU-bound and Disk-bound secondary tenants, respectively.

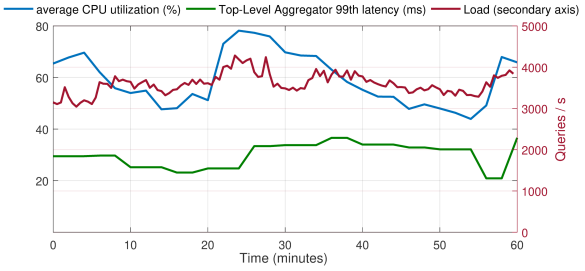


Figure 10: Production results for a cluster of 650 IndexServe machines colocated with a secondary running a machine learning training computation over 1 hour.

cycles available to the secondary, based on job progress information. Our experimental evaluation shows that managing CPU cycles per tenant severely impacts tail latency, and we manage the number of CPU cores of the secondary tenants dynamically instead.

TEMM [38] proposes a throttling-based management technique to configure CPU duty cycles and DVFS settings to meet SLOs. TEMM requires the latency SLOs of the primary, and assumes that the bottleneck is either the last-layer caches or off-chip bandwidth, whereas any of several other resources can be the bottleneck.

MIMP [36] considers tenants in a virtualized environment. It defines a new priority in the hypervisor scheduler to meet the performance requirements of the primary tenant, focusing on CPU-level interference.

Quasar [11] is a cluster manager that reduces resource over-provisioning and increases utilization while meeting quality of service constraints. It uses profiling information and collaborative filtering to infer the tenants' resource requirements. It additionally monitors service performance and adjusts allocations when target latencies are not met. In contrast, *PerfIso*, ensures some slack is always available to accommodate the bursty demands of the primary tenant without assuming explicit server-level performance requirements or tail latency targets.

Heracles [20] uses a feedback mechanism to adjust the secondary tenants' resources based on the tail latency of the primary tenant. Heracles needs the latency SLOs of the primary and exploits hardware mechanisms such as Intel's Cache Allocation Technology [2]. *Heracles* is complementary to this work since the knowledge of primary tenant SLOs and availability of specific hardware mechanisms limits wider deployment.

Jail [28] is Google's cache partitioning performance isolation mechanism. It incorporates Intel CMT and CAT [2] support into Linux cgroups, but allows only static partitioning of resources. In contrast our experiments show dynamic isolation techniques (such as *blind isolation*) provide more resources to the secondary while protecting the primary's tail latency. Additionally, data center machines span multiple hardware generations, not all of which supporting CAT.

RubikColoc [16] configures per-core DVFS to compensate for the overhead of multiplexing primary and secondary tenants on the same core. RubikColoc needs server-level latency SLOs and per-core DVFS support.

Elfen [34] provides CPU-level performance isolation for primary and secondary tenants running on the same core using Simultaneous Multi-Threading (SMT) technology available on modern CPUs. Effectively, the secondary is colocated with the primary on the same physical core only when the primary's measured performance is within its SLO. Elfen needs latency SLOs for the primary, OS support to query SMT-to-process mappings, and application-level instrumentation for the secondary.

BatchDB [23] is a database system that handles both OLTP and OLAP queries, providing good performance isolation for the former. Our approach instead focuses on the scenario where the tenants are distinct applications.

Leverich et al. [18] advocate using colocation to improve cluster throughput-per-TCO, and identify queuing delay, scheduling delay, and worker-thread load imbalance as the challenges in providing service-level QoS. They propose better cluster provisioning and custom OS schedulers to mitigate tenant interference. However, given the complexity of large commercial services, we

cannot make changes, neither at the hardware nor kernel level, which is why we adopt a "black-box" model.

CPI^2 [37] is a performance isolation mechanism which uses Clock Per Instruction (CPI) data to build probabilistic distribution models and to find stragglers (victims of resource interference). CPI^2 identifies the antagonists of latency-sensitive tasks by matching CPI patterns, and restricts their CPU cycle-share. We show that restricting CPU cores is significantly more effective in protecting primary tail latency.

HipsterCo [26] is a task scheduler for latency-sensitive workloads running on heterogeneous multi-core systems. HipsterCo colocates batch jobs with latency-sensitive workloads to increase resource utilization. HipsterCo uses reinforcement learning to build the CPU and DVFS configuration required to meet target SLOs. All remaining CPU cores are allocated to batch jobs. However, HipsterCo requires performance feedback from latency-sensitive workloads which is not available in our environment. Furthermore, we argue that for complex commercial latency-sensitive services, some buffer cores are required to prevent performance degradation.

Bubble-Up [24] and *Bubble-Flux* [33] focus on memory bandwidth and last-level cache interference as the main actors in colocation performance degradation. The former proposes a static profiling technique to accurately estimate the expected degradation, while the later uses an online approach, both assuming live performance information from the latency-sensitive service is available.

Zhang et al. [39] use historical resource utilization data and disk re-imaging patterns of tenants in task scheduling and data placement. The primary has resource-priority, meaning that a load-surge kills off secondary tasks. Thus, the scheduling algorithm places secondaries as to minimize the likelihood of termination. *Misra et al.* [25] improved upon this work by proposing a scalable distributed file system design which maximizes data availability for secondary tenants. Due to the highly spiking and unpredictable nature of the primary services we target, relying on historical data is insufficient to insure that performance guarantees are met.

Pisces [30] achieves fairness and per-tenant performance isolation in shared key-value storage. Pisces uses deficit-weighted-round-robin (DWRR) to schedule requests at server-level, thus mediating resource contention. In our case secondary tasks are batch jobs, and therefore do not lend themselves to request scheduling, so we only employ DWRR for I/O throttling.

2DFQ [22] proposes a new weighted fair queuing algorithm to ensure fairness for multi-tenant services that use thread pools inside a single process. This technique benefits the primary tenants and could reduce the burstiness of their execution, which complements *PerfIso* and potentially reduces the number of buffer cores required.

Alizadeh et. al [6] propose HULL — a system which leaves 'bandwidth headroom' to mitigate the problem of packet queuing in low-latency networked systems. This is similar in spirit to our non-work-conserving resource management approach, but focuses on avoiding network congestion rather than performance isolation.

8 Conclusions

Machines hosting large commercial latency-sensitive services are often underutilized because important services are provisioned for peak load as to meet business availably and fault-tolerance constraints.

Colocating batch jobs with latency-sensitive services is an important way of increasing utilization and data center efficiency, but comes with key challenges. Large latency-sensitive services are complex, and follow a layered architecture and therefore require short tail latencies. The diversity of commercial systems prevents us from using explicit knowledge of their latency targets, motivating us to adopt an approach in which we assume limited information about the primary tenant.

This paper presents the design and implementation of *PerfIso*, a performance isolation framework that makes use of idle resources to run batch jobs without affecting the primary tenant. It uses *CPU blind isolation* to meet the requirements of commercial large latency-sensitive services. The key insight is ensuring that the primary tenant always has idle cores available to accommodate its bursty workload, while allowing the secondary tenant to make progress.

We evaluate *PerfIso* experimentally on a single machine and on a cluster of machines using Bing IndexServe as the primary workload. We compare existing CPU isolation techniques (such as rate limiting and static core affinitization) to our approach, and we find that under the latter the 99^{th} percentile of the tail latency values remain largely unchanged compared to running standalone. *PerfIso* allows compute-intensive batch jobs to use up to 47% of CPU cycles for off-peak loads which would have otherwise remained idle.

Acknowledgements We thank our anonymous reviewers for their valuable feedback and advice. We thank our colleagues in Microsoft Windows: Danyu Zhu, Kai Hsu, Deepu Thomas, Mehmet Iyigun, Milos Kralj and Eugene Bak, as well as Ricardo Bianchini (Microsoft Research), Íñigo Goiri (Microsoft Research), Willy Zwaenepoel (EPFL), Marcus Fountoura (Microsoft Azure), Ramashis Das (Microsoft Bing), Kukjin Lee (Microsoft Research).

References

[1] Hadoop. http://hadoop.apache.org.

[2] Intel CAT. https://www.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html.

[3] Windows Job Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/hh684161(v=vs.85).aspx.

[4] Cgroups, 2014. http://en.wikipedia.org/wiki/Cgroups.

[5] DiskSPD, 2017. https://github.com/Microsoft/diskspd.

[6] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 19–19.

[7] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., ET AL. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[8] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[9] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[10] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[11] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 127–144.

[12] DOUCEUR, J. R., AND BOLOSKY, W. J. Progress-based regulation of low-importance processes. In *In Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (1999), ACM Press, pp. 247–260.

[13] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA* (2006), vol. 33, pp. 10–17.

[14] ISARD, M. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (Apr. 2007), 60–67.

[15] JEON, M., HE, Y., KIM, H., ELNIKETY, S., RIXNER, S., AND COX, A. L. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, pp. 129–141.

[16] KASTURE, H., BARTOLINI, D. B., BECKMANN, N., AND SANCHEZ, D. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), ACM, pp. 598–610.

[17] KIM, S., HE, Y., HWANG, S.-W., ELNIKETY, S., AND CHOI, S. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining* (2015), ACM, pp. 7–16.

[18] LEVERICH, J., AND KOZYRAKIS, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 4.

[19] LI, T., BAUMBERGER, D., AND HAHN, S. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 65–74.

[20] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Improving resource efficiency at scale with Heracles. *ACM Transactions on Computer Systems (TOCS) 34*, 2 (2016), 6.

[21] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 1.

[22] MACE, J., BODIK, P., MUSUVATHI, M., FONSECA, R., AND VARADARAJAN, K. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 144–159.

[23] MAKRESHANSKI, D., GICEVA, J., BARTHELS, C., AND ALONSO, G. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 37–50.

[24] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 248–259.

[25] MISRA, P. A., GOIRI, I., KACE, J., AND BIANCHINI, R. Scaling distributed file systems in resource-harvesting datacenters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 799–811.

[26] NISHTALA, R., CARPENTER, P., PETRUCCI, V., AND MARTORELL, X. Hipster: Hybrid task manager for latency-critical cloud workloads. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 409–420.

[27] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., CHUN, B.-G., AND ICSI, V. Making sense of performance in data analytics frameworks. In *NSDI* (2015), vol. 15, pp. 293–307.

[28] ROHIT, J., AND DAVID, L. CAT at scale: Deploying cache isolation in a mixed workload environment. LinuxCon + ContainerCon North America, August 2016.

[29] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. In *velocity web performance and operations conference* (2009).

[30] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance isolation and fairness for multi-tenant cloud storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 349–362.

[31] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[32] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[33] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 607–618.

[34] YANG, X., BLACKBURN, S. M., AND MCKINLEY, K. S. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference* (2016), pp. 309–322.

[35] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., ET AL. Apache Spark: A unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[36] ZHANG, W., RAJASEKARAN, S., DUAN, S., WOOD, T., AND ZHUY, M. Minimizing interference and maximizing progress for Hadoop virtual machines. *ACM SIGMETRICS Performance Evaluation Review 42*, 4 (2015), 62–71.

[37] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 379–391.

[38] ZHANG, X., ZHONG, R., DWARKADAS, S., AND SHEN, K. A flexible framework for throttling-enabled multicore management (TEMM). In *Parallel Processing (ICPP), 2012 41st International Conference on* (2012), IEEE, pp. 389–398.

[39] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 755–770.